

Índice

1. Introducción	2
1.1. Planteamiento del problema	2
1.2. Motivación	2
1.3. Objetivos	2
2. Marco Teórico	3
2.1. Clases y objetos	3
2.2. Constructores	4
2.3. Encapsulación, Getters y Setters	6
2.4. Herencia	8
2.5. Polimorfismo	10
2.6. Clase abstracta	11
2.7. Archivo y dart:io	13
2.8. Manejo de errores	16
2.9. Programación asíncrona: futuros, asíncrono, espera	16
2.9.1. Future	17
2.9.2. Trabajar con futuros: async y await	17
3. Desarrollo	19
3.1. Archivos	19
3.2. Hilos	22
4. Resultados	23
5. Conclusiones	24

1. Introducción

En el estudio de la programación no solo se requiere de la comprensión de los lenguajes y sus sintaxis, sino también la capacidad de analizar cómo los distintos conceptos se integran en el diseño y funcionamiento de un sistema. En este sentido, la práctica sobre archivos, hilos y patrones de diseño busca ofrecer una visión integral de tres componentes esenciales en el desarrollo de software, apoyándose en representaciones gráficas mediante diagramas UML estáticos y dinámicos y en la interpretación crítica de ejemplos concretos.

1.1. Planteamiento del problema

En el desarrollo de software, los estudiantes se deben enfrentar al reto de comprender y aplicar conceptos que van más allá de la programación básica. El manejo de archivos permite la persistencia y organización de datos; los hilos introducen la complejidad de la ejecución concurrente y la sincronización de procesos; mientras que los patrones de diseño ofrecen soluciones estructuradas a problemas recurrentes en la arquitectura de sistemas. Sin embargo, la dificultad radica en integrar estos elementos de manera coherente, comprendiendo no solo su implementación técnica, sino también su impacto en la calidad, eficiencia y mantenibilidad del código.

1.2. Motivación

La práctica surge de la necesidad de fortalecer las competencias en programación avanzada mediante ejemplos concretos que permitan visualizar cómo los conceptos teóricos se aplican en situaciones reales. El uso de diagramas UML estáticos y dinámicos se convierte en una herramienta clave para representar tanto la estructura como el comportamiento del sistema, facilitando la interpretación y el análisis crítico. Además, trabajar con material audiovisual proporcionado por el docente asegura un aprendizaje contextualizado, donde los estudiantes pueden relacionar la teoría con casos prácticos y desarrollar una visión más integral del diseño de software.

1.3. Objetivos

Se realizara un reporte con los siguientes temas:

- Explicación de la aplicación del tema de Archivos en el ejemplo principal del video.
- Explicación de la aplicación del tema de Hilos en los ejemplos del video.
- Explicación de la aplicación del tema de Patrones de Diseño en el ejemplo principal del video.

Con base en los códigos, se reportara lo siguiente:

- Diagramas UML estático.
- Diagramas UML dinámico.
- Interpretación de conceptos aplicados en el código.

2. Marco Teórico

2.1. Clases y objetos

Dart es un lenguaje orientado a objetos con clases y herencia basada en mixins. Cada objeto es una instancia de una clase, y todas las clases, excepto Null, descienden de Object. La herencia basada en mixins significa que, aunque cada clase (excepto la clase superior, ¿Object?) tiene exactamente una superclase, el cuerpo de una clase se puede reutilizar en múltiples jerarquías de clases. Los métodos de extensión son una forma de añadir funcionalidad a una clase sin cambiarla ni crear una subclase. Los modificadores de clase permiten controlar cómo las bibliotecas pueden subtipificar una clase.[?]

En la programación orientada a objetos, un objeto es una unidad autocontenida de código y datos. Los objetos se crean a partir de plantillas llamadas clases. Un objeto se compone de propiedades (variables) y métodos (funciones). Un objeto es una instancia de una clase.[?]

- Utilizando miembros de clase:

Los objetos tienen miembros que consisten en funciones y datos (métodos y variables de instancia, respectivamente). Al llamar a un método, se invoca sobre un objeto: el método tiene acceso a las funciones y datos de ese objeto. Utilice un punto (.) para referirse a una variable de instancia o método:

```
1 var p = Point(2, 2);
2
3 // Get the value of y.
4 assert(p.y == 2);
5
6 // Invoke distanceTo() on p.
7 double distance = p.distanceTo(Point(4, 4));
```

Utilice ?. en su lugar . para evitar una excepción cuando el operando más a la izquierda sea nulo:

```
1 // If p is non-null, set a variable equal to its y value.
2 var a = p?.y;
```

- Utilizando constructores Puedes crear un objeto usando un constructor. Los nombres de los constructores pueden ser 'init' ClassName o 'init'. Por ejemplo, el siguiente código crea objetos usando los constructores 'init' y 'init': ClassName.identifierPointPoint()Point.fromJson().[?]

```
1 var p1 = Point(2, 2);
2 var p2 = Point.fromJson({'x': 1, 'y': 2});
```

El siguiente código tiene el mismo efecto, pero utiliza la new palabra clave opcional antes del nombre del constructor:

```
1 var p1 = new Point(2, 2);
2 var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

- Variables y métodos de clase

utilice la static palabra clave para implementar variables y métodos de toda la clase.

```
1 class Queue {
2     static const initialCapacity = 16;
3
4 }
5
6 void main() {
7     assert(Queue.initialCapacity == 16);
8 }
```

- métodos estáticos

Los métodos estáticos (métodos de clase) no operan sobre una instancia y, por lo tanto, no tienen acceso a ella `this`. Sin embargo, sí tienen acceso a las variables estáticas. Como muestra el siguiente ejemplo, los métodos estáticos se invocan directamente en una clase.[?]

```
1 import 'dart:math';
2
3 class Point {
4     double x, y;
5     Point(this.x, this.y);
6
7     static double distanceBetween(Point a, Point b) {
8         var dx = a.x - b.x;
9         var dy = a.y - b.y;
10        return sqrt(dx * dx + dy * dy);
11    }
12 }
13
14 void main() {
15     var a = Point(2, 2);
16     var b = Point(4, 4);
17     var distance = Point.distanceBetween(a, b);
18     assert(2.8 < distance && distance < 2.9);
19     print(distance);
20 }
```

2.2. Constructores

Los constructores son funciones especiales que crean instancias de clases.

Dart implementa muchos tipos de constructores. Excepto los constructores por defecto, estas funciones utilizan el mismo nombre que su clase.[?]

1. constructores generativos Para instanciar una clase, utilice un constructor generativo.

```
1 class Point {
2     // Instance variables to hold the coordinates of the point.
3     double x;
```

```

4   double y;
5
6   // Generative constructor with initializing formal parameters:
7   Point(this.x, this.y);
8 }

```

2. Constructores por defecto

Si no declaras un constructor, Dart utiliza el constructor predeterminado. El constructor predeterminado es un constructor generativo sin argumentos ni nombre.[?]

3. Constructores con nombre

Utilice un constructor con nombre para implementar múltiples constructores para una clase o para proporcionar mayor claridad:

```

1  vconst double xOrigin = 0;
2  const double yOrigin = 0;
3
4  class Point {
5      final double x;
6      final double y;
7
8      // Sets the x and y instance variables
9      // before the constructor body runs.
10     Point(this.x, this.y);
11
12     // Named constructor
13     Point.origin() : x = xOrigin, y = yOrigin;
14 }

```

Una subclase no hereda el constructor con nombre de la superclase. Para crear una subclase con un constructor con nombre definido en la superclase, implemente dicho constructor en la subclase.

4. Constructores constantes

Si tu clase produce objetos que no cambian, haz que estos objetos sean constantes en tiempo de compilación. Para ello, define un constconstructor con todas las variables de instancia establecidas como final.[?]

```

1  class ImmutablePoint {
2      static const ImmutablePoint origin = ImmutablePoint(0, 0);
3
4      final double x, y;
5
6      const ImmutablePoint(this.x, this.y);
7  }

```

Los constructores constantes no siempre crean constantes. Pueden invocarse fuera de un constcontexto. Para obtener más información, consulte la sección sobre el uso de constructores

5. Constructores de redireccionamiento

Un constructor puede redirigir a otro constructor de la misma clase. Un constructor que redirige tiene un cuerpo vacío. El constructor utiliza `this` en lugar del nombre de la clase después de dos puntos (`:`).

```
1 class Point {
2     double x, y;
3
4     // The main constructor for this class.
5     Point(this.x, this.y);
6
7     // Delegates to the main constructor.
8     Point.alongXAxis(double x) : this(x, 0);
9 }
```

2.3. Encapsulacion, Getters y Setters

En Dart, la encapsulación consiste en ocultar datos dentro de una biblioteca, protegiéndolos de factores externos. Esto ayuda a controlar el programa y evita que se vuelva demasiado complejo.[?]

La encapsulación se puede lograr mediante:

- Declarar las propiedades de la clase como privadas utilizando guion bajo (`_`).
- Proporcionar métodos públicos `getter` y `setter` para acceder y actualizar el valor de la propiedad privada.

Nota: Dart no admite palabras clave como `'public'`, `'private'` y `'protected'`. Dart utiliza `"_"` (guion bajo) para indicar que una propiedad o método es privado. Los métodos `getter` y `setter` se utilizan para acceder y actualizar el valor de una propiedad privada. Los métodos `getter` se utilizan para acceder al valor de una propiedad privada. Los métodos `setter` se utilizan para actualizar el valor de una propiedad privada.[?]

Ejemplo 1: En este ejemplo, crearemos una clase llamada **Employee**. Esta clase tendrá dos propiedades privadas: `_id` y `_name`. También crearemos dos métodos públicos, `getId()` y `getName()`, para acceder a las propiedades privadas. Asimismo, crearemos dos métodos públicos, `setId()` y `setName()`, para actualizar las propiedades privadas.[?]

```
1 class Employee {
2     // Private properties
3     int? _id;
4     String? _name;
5
6     // Getter method to access private property _id
7     int getId() {
8         return _id!;
9     }
10    // Getter method to access private property _name
11    String getName() {
12        return _name!;
13    }
14    // Setter method to update private property _id
```

```

15     void setId(int id) {
16         this._id = id;
17     }
18     // Setter method to update private property _name
19     void setName(String name) {
20         this._name = name;
21     }
22
23 }
24
25 void main() {
26     // Create an object of Employee class
27     Employee emp = new Employee();
28     // setting values to the object using setter
29     emp.setId(1);
30     emp.setName("John");
31
32     // Retrieve the values of the object using getter
33     print("Id: ${emp.getId()}");
34     print("Name: ${emp.getName()}");
35 }

```

En Dart, puedes controlar el acceso a las propiedades e implementar la encapsulación mediante el uso de propiedades de solo lectura. Para ello, añade la palabra clave ‘final’ antes de la declaración de la propiedad. De esta forma, solo podrás acceder a su valor, pero no modificarlo.[?]

```

1 class Student {
2     final _schoolname = "ABC School";
3
4     String getSchoolName() {
5         return _schoolname;
6     }
7 }
8
9 void main() {
10     var student = Student();
11     print(student.getSchoolName());
12     // This is not possible
13     //student._schoolname = "XYZ School";
14 }

```

Puedes crear métodos getter y setter usando las palabras clave **get** y **set**. En el siguiente ejemplo, hemos creado una clase llamada **Vehicle**. Esta clase tiene dos propiedades privadas: `_model` y `_year`. También hemos creado dos métodos getter y setter para cada propiedad. Estos métodos se llaman `model` y `year`, y se utilizan para acceder y actualizar el valor de las propiedades privadas.[?]

```

1 class Vehicle {
2     String _model;
3     int _year;

```

```

4
5 // Getter method
6 String get model => _model;
7
8 // Setter method
9 set model(String model) => _model = model;
10
11 // Getter method
12 int get year => _year;
13
14 // Setter method
15 set year(int year) => _year = year;
16 }
17
18 void main() {
19     var vehicle = Vehicle();
20     vehicle.model = "Toyota";
21     vehicle.year = 2019;
22     print(vehicle.model);
23     print(vehicle.year);
24 }

```

¿Por qué es importante la encapsulación?

1. Ocultación de datos : La encapsulación oculta los datos del exterior. Impide que el código fuera de la clase acceda a ellos. Esto se conoce como ocultación de datos.[?]
2. Facilidad de prueba : La encapsulación permite probar la clase de forma aislada. Esto permite probar la clase sin necesidad de probar el código fuera de ella.[?]
3. Flexibilidad : La encapsulación permite cambiar la implementación de la clase sin afectar al código que se encuentra fuera de la clase.[?]
4. Seguridad : La encapsulación permite restringir el acceso a los miembros de la clase. Esto permite limitar el acceso a los miembros de la clase desde el código externo a la biblioteca.[?]

2.4. Herencia

La herencia es la compartición de comportamiento entre dos clases. Permite definir una clase que extiende la funcionalidad de otra. La palabra clave ‘extend’ se utiliza para heredar de la clase padre.[?]

Cuando se utiliza la herencia, siempre se crea una relación ^{es} un/una ^{entre} la clase padre y la clase hija, como por ejemplo: Estudiante es una Persona , Camión es un Vehículo , Vaca es un Animal , etc.

Dart admite herencia simple, lo que significa que una clase solo puede heredar de una única clase. Dart no admite herencia múltiple, lo que significa que una clase no puede heredar de varias clases.[?]

```

1 class ParentClass {
2     // Parent class code
3 }
4
5 class ChildClass extends ParentClass {

```



```
6 // Child class code
7 }
```

En esta sintaxis, ParentClass es la superclase y ChildClass es la subclase. ChildClass hereda las propiedades y métodos de ParentClass .

Términos:

- Clase padre: La clase cuyas propiedades y métodos hereda otra clase se denomina clase padre. También se la conoce como clase base o superclase.[?]
- Clase hija: La clase que hereda las propiedades y métodos de otra clase se denomina clase hija. También se la conoce como clase derivada o subclase.[?]

Ejemplo 1: En este ejemplo, crearemos una clase Persona y luego crearemos una clase Estudiante que hereda las propiedades y métodos de la clase Persona

```
1 class Person {
2     // Properties
3     String? name;
4     int? age;
5
6     // Method
7     void display() {
8         print("Name: $name");
9         print("Age: $age");
10    }
11 }
12 // Here In student class , we are extending the
13 // properties and methods of the Person class
14 class Student extends Person {
15     // Fields
16     String? schoolName;
17     String? schoolAddress;
18
19     // Method
20     void displaySchoolInfo() {
21         print("School Name: $schoolName");
22         print("School Address: $schoolAddress");
23     }
24 }
25
26 void main() {
27     // Creating an object of the Student class
28     var student = Student();
29     student.name = "John";
30     student.age = 20;
31     student.schoolName = "ABC School";
32     student.schoolAddress = "New York";
33     student.display();
34     student.displaySchoolInfo();
35 }
```

Ventajas de la herencia:

- Promueve la reutilización del código y reduce el código redundante
- Ayuda a diseñar un programa de mejor manera.
- Simplifica y limpia el código, y ahorra tiempo y dinero en mantenimiento.
- Facilita la creación de bibliotecas de clases.
- Puede utilizarse para imponer una interfaz estándar a todas las clases hijas.

2.5. Polimorfismo

Poli significa muchos y morfo significa formas . El polimorfismo es la capacidad de un objeto para adoptar muchas formas. Como seres humanos, tenemos la capacidad de adoptar muchas formas. Podemos ser estudiantes, profesores, padres, amigos, etc. De manera similar, en la programación orientada a objetos, el polimorfismo es la capacidad de un objeto para adoptar muchas formas.[?]

La sobrescritura de métodos es una técnica que permite crear un método en la clase hija con el mismo nombre que el método en la clase padre. El método en la clase hija sobrescribe el método en la clase padre.[?]

```
1 class ParentClass{
2   void functionName(){
3   }
4 }
5 class ChildClass extends ParentClass{
6   @override
7   void functionName(){
8   }
9 }
```

Ejemplo 1: Polimorfismo mediante la sobrescritura de métodos en Dart

En el siguiente ejemplo, existe una clase llamada Animal con un método llamado eat() . El método eat() se redefine en la clase hija llamada Dog .

```
1 class Animal {
2   void eat() {
3     print("Animal is eating");
4   }
5 }
6
7 class Dog extends Animal {
8   @override
9   void eat() {
10    print("Dog is eating");
11  }
12 }
13
14 void main() {
15   Animal animal = Animal();
16   animal.eat();
17 }
```

```

17
18   Dog dog = Dog();
19   dog.eat();
20 }

```

Ventaja de usar polimorfismo:

1. Las subclases pueden anular el comportamiento de la clase padre.
2. Nos permite escribir código más flexible y reutilizable.

2.6. Clase abstracta

Las clases abstractas son clases que no se pueden inicializar. Se utilizan para definir el comportamiento de una clase que puede ser heredado por otras clases. Una clase abstracta se declara utilizando la palabra clave `abstract`.^[?]

```

1  abstract class ClassName {
2      //Body of abstract class
3
4      method1();
5      method2();
6  }

```

Un método abstracto es un método que se declara sin implementación. Se declara con un punto y coma (;) en lugar del cuerpo del método.^[?]

```

1  abstract class ClassName {
2      //Body of abstract class
3      method1();
4      method2();
5  }

```

Las subclases de una clase abstracta deben implementar todos los métodos abstractos de la clase abstracta. Se utiliza para lograr la abstracción en el lenguaje de programación Dart.^[?]

Ejemplo 1: En el siguiente ejemplo, hay una clase abstracta Vehículo con dos métodos abstractos `start()` y `stop()`. Las subclases `Auto` y `Bicicleta` implementan los métodos abstractos y los sobrescriben para imprimir el mensaje..^[?]

```

1  abstract class Vehicle {
2      // Abstract method
3      void start();
4      // Abstract method
5      void stop();
6  }
7
8  class Car extends Vehicle {
9      // Implementation of start()
10     @override
11     void start() {

```

```

12     print('Car started ');
13 }
14
15 // Implementation of stop()
16 @override
17 void stop() {
18     print('Car stopped ');
19 }
20 }
21
22 class Bike extends Vehicle {
23     // Implementation of start()
24     @override
25     void start() {
26         print('Bike started ');
27     }
28
29     // Implementation of stop()
30     @override
31     void stop() {
32         print('Bike stopped ');
33     }
34 }
35
36 void main() {
37     Car car = Car();
38     car.start();
39     car.stop();
40
41     Bike bike = Bike();
42     bike.start();
43     bike.stop();
44 }

```

Nota: La clase abstracta se utiliza para definir el comportamiento de una clase que puede ser heredada por otras clases. Se puede definir un método abstracto dentro de una clase abstracta.[?]

No se puede crear un objeto de una clase abstracta. Sin embargo, se puede definir un constructor en una clase abstracta. El constructor de una clase abstracta se invoca cuando se crea un objeto de una subclase.[?]

Ejemplo 2: En el siguiente ejemplo, hay una clase abstracta Bank con un constructor que toma dos parámetros: nombre y tasa. Hay un método abstracto interest(). Las subclases SBI e ICICI implementan el método abstracto y lo sobrescriben para imprimir la tasa de interés.

```

1 abstract class Bank {
2     String name;
3     double rate;
4
5     // Constructor
6     Bank(this.name, this.rate);

```

```

7
8 // Abstract method
9 void interest ();
10
11 //Non-Abstract method: It have an implementation
12 void display () {
13     print ('Bank Name: $name ');
14 }
15 }
16
17 class SBI extends Bank {
18     // Constructor
19     SBI(String name, double rate) : super(name, rate);
20
21     // Implementation of interest ()
22     @override
23     void interest () {
24         print ('The rate of interest of SBI is $rate ');
25     }
26 }
27
28 class ICICI extends Bank {
29     // Constructor
30     ICICI(String name, double rate) : super(name, rate);
31
32     // Implementation of interest ()
33     @override
34     void interest () {
35         print ('The rate of interest of ICICI is $rate ');
36     }
37 }
38
39 void main () {
40     SBI sbi = SBI ('SBI', 8.4);
41     ICICI icici = ICICI ('ICICI', 7.3);
42
43     sbi.interest ();
44     icici.interest ();
45     icici.display ();
46 }

```

2.7. Archivo y dart:io

La biblioteca dart:io proporciona API para gestionar archivos, directorios, procesos, sockets, Web-Sockets y clientes y servidores HTTP. [?]

En general, la biblioteca dart:io implementa y promueve una API asíncrona. Los métodos síncronos pueden bloquear fácilmente una aplicación, dificultando su escalabilidad. Por lo tanto, la mayoría de las operaciones devuelven resultados mediante objetos Future o Stream, un patrón común en plataformas de servidor modernas como Node.js.[?]

Los pocos métodos sincrónicos de la biblioteca `dart:io` están claramente marcados con el sufijo `Sync` en el nombre del método. No se tratan aquí los métodos sincrónicos.[?]

```
1 import 'dart:io';
```

A `File` contiene una ruta donde se pueden realizar operaciones. Puedes obtener el directorio padre del archivo usando `parent`, una propiedad heredada de `FileSystemEntity`.

Cree un nuevo `File` objeto con una ruta para acceder al archivo especificado en el sistema de archivos desde su programa.

```
1 var myFile = File('file.txt');
```

La `File` clase contiene métodos para manipular archivos y su contenido. Con los métodos de esta clase, puede abrir y cerrar archivos, leerlos y escribir en ellos, crearlos y eliminarlos, y comprobar su existencia.[?]

Al leer o escribir un archivo, puede utilizar secuencias (con `openRead`), operaciones de acceso aleatorio (con `open`) o métodos convenientes como `readAsString`. [?]

La mayoría de los métodos de esta clase se ejecutan en pares síncronos y asíncronos; por ejemplo, `readAsString` y `readAsStringSync`. A menos que tenga una razón específica para usar la versión síncrona de un método, prefiera la versión asíncrona para evitar bloquear su programa. Si `path` es un enlace simbólico, en lugar de un archivo, entonces los métodos de `File` operan en el destino final del enlace, excepto `delete` y `deleteSync`, que operan en el enlace.

- Leer desde un archivo: El siguiente ejemplo de código lee todo el contenido de un archivo como una cadena utilizando el método asíncrono `readAsString`

```
1 import 'dart:async';
2 import 'dart:io';
3
4 void main() {
5   File('file.txt').readAsString().then((String contents) {
6     print(contents);
7   });
8 }
```

Una forma más flexible y útil de leer un archivo es con un `Stream`. Abra el archivo con `openRead`, que devuelve un `stream` que proporciona los datos del archivo como fragmentos de bytes. Lea el `stream` para procesar el contenido del archivo cuando esté disponible. Puede usar varios transformadores sucesivamente para manipular el contenido del archivo al formato requerido o para prepararlo para la salida.[?]

Es posible que desee utilizar una secuencia para leer archivos grandes, para manipular los datos con transformadores o para lograr compatibilidad con otra API, como `WebSockets`. [?]

```
1 import 'dart:io';
2 import 'dart:convert';
3 import 'dart:async';
4
5 void main() async {
6   final file = File('file.txt');
7   Stream<String> lines = file.openRead()
```

```

8      .transform(utf8.decoder)          // Decode bytes to UTF-8.
9      .transform(LineSplitter());      // Convert stream to individual lines.
10     try {
11       await for (var line in lines) {
12         print('$line: ${line.length} characters ');
13       }
14       print('File is now closed. ');
15     } catch (e) {
16       print('Error: $e');
17     }
18   }

```

- Escribir en un archivo: Para escribir una cadena en un archivo, utilice el método `writeAsString`

```

1 import 'dart:io';
2
3 void main() async {
4   final filename = 'file.txt';
5   var file = await File(filename).writeAsString('some content');
6   // Do something with the file.
7 }

```

También puedes escribir en un archivo usando un `Stream`. Abre el archivo con `openWrite`, que devuelve un `IOSink` donde puedes escribir datos. Asegúrate de cerrar el `IOSink` con el método `IOSink.close`.

```

1 import 'dart:io';
2
3 void main() async {
4   var file = File('file.txt');
5   var sink = file.openWrite();
6   sink.write('FILE ACCESSED ${DateTime.now()}\n');
7   await sink.flush();
8
9   // Close the IOSink to free system resources.
10  await sink.close();
11 }

```

Para evitar bloqueos involuntarios del programa, varios métodos son asíncronos y devuelven un `Future`. Por ejemplo, el método `length`, que obtiene la longitud de un archivo, devuelve un `Future`. Espere a que el future obtenga el resultado cuando esté listo.

```

1 import 'dart:io';
2
3 void main() async {
4   final file = File('file.txt');
5
6   var length = await file.length();
7   print(length);
8 }

```

2.8. Manejo de errores

Su código Dart puede generar y capturar excepciones. Las excepciones son errores que indican que ocurrió algo inesperado. Si no se captura la excepción, el aislamiento que la generó se suspende y, por lo general, el aislamiento y su programa se cierran.[?]

A diferencia de Java, todas las excepciones de Dart son excepciones no comprobadas. Los métodos no declaran qué excepciones podrían lanzar, y no es necesario capturar ninguna.[?]

Dart proporciona tipos `Exception` y `Error`, así como numerosos subtipos predefinidos. Por supuesto, puede definir sus propias excepciones. Sin embargo, los programas Dart pueden lanzar cualquier objeto no nulo como excepción, no solo objetos de excepción y error.[?]

```
1 throw FormatException('Expected at least 1 section');
```

Capturar una excepción impide que se propague (a menos que se vuelva a lanzar). Capturar una excepción permite gestionarla:

```
1 try {  
2   breedMoreLlamas();  
3 } on OutOfLlamasException {  
4   buyMoreLlamas();  
5 }
```

Para gestionar código que puede generar más de un tipo de excepción, se pueden especificar varias cláusulas `catch`. La primera cláusula `catch` que coincida con el tipo del objeto generado gestiona la excepción. Si la cláusula `catch` no especifica un tipo, puede gestionar cualquier tipo de objeto generado: [?]

```
1 try {  
2   breedMoreLlamas();  
3 } on OutOfLlamasException {  
4   // A specific exception  
5   buyMoreLlamas();  
6 } on Exception catch (e) {  
7   // Anything else that is an exception  
8   print('Unknown exception: $e');  
9 } catch (e) {  
10  // No specified type, handles all  
11  print('Something really unknown: $e');  
12 }
```

Como se muestra en el código anterior, puede usar uno de los dos `on`, `catch`o ambos. Úselo `on` cuando necesite especificar el tipo de excepción. Úselo `catch` cuando su manejador de excepciones necesite el objeto de excepción.[?]

Puedes especificar uno o dos parámetros para `catch()`. El primero es la excepción lanzada y el segundo es el seguimiento de la pila (un `StackTrace` objeto).

2.9. Programación asíncrona: futuros, asíncrono, espera

Las operaciones asíncronas permiten que su programa complete su trabajo mientras espera que finalice otra operación. Estas son algunas operaciones asíncronas comunes:

Obteniendo datos a través de una red. Escribiendo en una base de datos. Leyendo datos de un archivo. Estos cálculos asíncronos suelen proporcionar su resultado como un `Future`, si el resultado tiene varias partes, como un `Stream`. Estos cálculos introducen asincronía en un programa. Para adaptar esta asincronía inicial, otras funciones Dart simples también deben volverse asíncronas.

Para interactuar con estos resultados asíncronos, puede usar las palabras clave `async` y `await`. La mayoría de las funciones asíncronas son simplemente funciones asíncronas de Dart que dependen, posiblemente en el fondo, de un cálculo inherentemente asíncrono.

2.9.1. Future

Un futuro (con "f" minúscula) es una instancia de la clase `Future` (con "F" mayúscula). Un futuro representa el resultado de una operación asíncrona y puede tener dos estados: incompleto o completo.

Incompleto

Al llamar a una función asíncrona, esta devuelve un futuro incompleto. Este futuro espera a que la operación asíncrona de la función finalice o arroje un error.

Terminado

Si la operación asíncrona se realiza correctamente, el futuro se completa con un valor. De lo contrario, se completa con un error.

Un futuro de tipo `FutureT` se completa con un valor de tipo `T`. Por ejemplo, un futuro con tipo `FutureString` produce un valor de cadena. Si un futuro no produce un valor utilizable, entonces su tipo es `FutureVoid`.

Completando con un error

Si la operación asíncrona realizada por la función falla por cualquier motivo, el futuro se completa con un error.

```
1 Future<void> fetchUserOrder() {
2   // Imagine that this function is fetching user info from another service or database.
3   return Future.delayed(const Duration(seconds: 2), () => print('Large Latte'));
4 }
5
6 void main() {
7   fetchUserOrder();
8   print('Fetching user order...');
9 }
```

2.9.2. Trabajar con futuros: `async` y `await`

Las palabras clave `async` y `await` proporcionan una forma declarativa de definir funciones asíncronas y utilizar sus resultados. Recuerde estas dos pautas básicas al usar `async` y `await`: [?] Para definir una función asíncrona, agregue `async` antes del cuerpo de la función: La palabra clave `await` sólo funciona en funciones asíncronas.[?] A continuación se muestra un ejemplo que convierte `main()` de una función sincrónica a una asíncrona.[?]

Primero, agregue la palabra clave `async` antes del cuerpo de la función:

```
1 void main() async {
```

Si la función tiene un tipo de retorno declarado, actualízelo a FutureT, donde T es el tipo del valor que devuelve la función. Si la función no devuelve un valor explícitamente, el tipo de retorno es Futurevoid:

```
1 Future<void> main() async { }
```

Ejemplo: funciones sincrónicas

```
1 String createOrderMessage() {  
2   var order = fetchUserOrder();  
3   return 'Your order is: $order';  
4 }  
5  
6 Future<String> fetchUserOrder() =>  
7   // Imagine that this function is  
8   // more complex and slow.  
9   Future.delayed(const Duration(seconds: 2), () => 'Large Latte');  
10  
11 void main() {  
12   print('Fetching user order...');  
13   print(createOrderMessage());  
14 }
```

Ejemplo: funciones asincrónicas

```
1 Future<String> createOrderMessage() async {  
2   var order = await fetchUserOrder();  
3   return 'Your order is: $order';  
4 }  
5  
6 Future<String> fetchUserOrder() =>  
7   // Imagine that this function is  
8   // more complex and slow.  
9   Future.delayed(const Duration(seconds: 2), () => 'Large Latte');  
10  
11 Future<void> main() async {  
12   print('Fetching user order...');  
13   print(await createOrderMessage());  
14 }
```

3. Desarrollo

A continuación se pasará a explicar detalladamente cada parte fundamental de los temas en el siguiente orden:

3.1. Archivos

El programa proporcionado presenta un menú interactivo que permite al usuario realizar tres operaciones fundamentales sobre archivos de texto: crear y escribir, leer, y sobrescribir. Cada una de estas acciones está encapsulada en funciones específicas que reflejan buenas prácticas de modularidad y control de flujo.

- 1.Crear y escribir archivos: La función `crearYEscribirArchivo()` permite al usuario crear un archivo nuevo y guardar texto línea por línea. Este proceso refleja el concepto de persistencia de datos y buffer manual.

```
1 File archivo = File(nombreArchivo);
2 archivo.writeAsStringSync(lineas.join('\n'));
```

- Se crea una instancia de `File`, que representa un archivo en el sistema.
- `writeAsStringSync` escribe el contenido de forma sincrónica, garantizando que el archivo se guarde antes de continuar.
- El uso de `lineas.join('\n')` simula un buffer: se acumulan las líneas en memoria antes de escribirlas.

- 2.Leer archivos existentes: La función `_leerArchivoExistente()` permite recuperar el contenido de un archivo previamente creado. Aquí se aplica el concepto de extracción de datos persistentes.

```
1 if (!archivo.existsSync()) {
2     print('\nEl archivo no existe.\n');
3     return;
4 }
5
6 String contenido = archivo.readAsStringSync();
```

- Se verifica la existencia del archivo con `existsSync()`, lo que evita errores de lectura.
- `readAsStringSync()` carga todo el contenido en una sola operación, útil para archivos pequeños.

- 3.Sobrescribir archivos: La función `_sobrescribirArchivo()` introduce el concepto de modificación destructiva. Antes de sobrescribir, se solicita confirmación explícita del usuario.

```
1 if (confirmacion == null || confirmacion.toUpperCase() != 'SI') {
2     print('\nOperacion cancelada.\n');
3     return;
4 }
```

- Se protege la integridad del archivo solicitando confirmación.
- Si el usuario acepta, se recopila nuevo contenido y se usa `writeAsStringSync` para reemplazar el archivo.

- Conexión entre funciones: Todas las funciones comparten un patrón de interacción:

```
1 String? entrada = stdin.readLineSync();
```

- Se usa entrada estándar para interactuar con el usuario, lo que permite construir un flujo conversacional.
- Este patrón refuerza la idea de que el usuario es parte activa del proceso de persistencia.

El diagrama estático que representa el código Archivos es el siguiente:

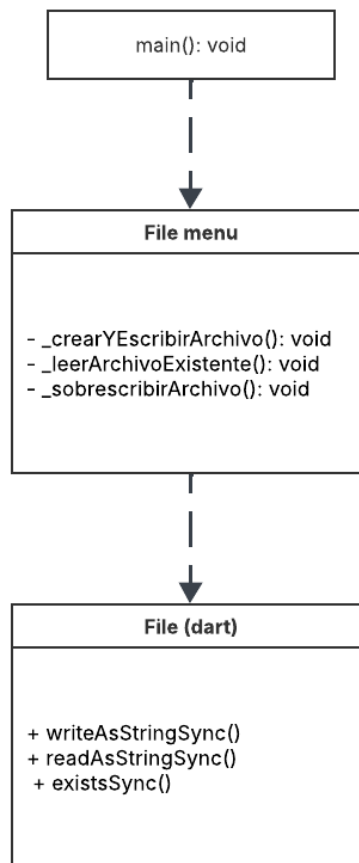


Figura 1: Diagrama estático / ARCHIVOS

El diagrama estático representa la estructura del programa en Dart que gestiona operaciones con archivos mediante un menú interactivo; la función principal `main()` actúa como punto de entrada y dirige el flujo hacia tres funciones privadas agrupadas conceptualmente en la clase `File menu`: `_crearYEscribirArchivo()`, `_leerArchivoExistente()` y `_sobrescribirArchivo()`, cada una encargada de una operación específica sobre archivos de texto; estas funciones interactúan directamente con la clase `File` de Dart, utilizando sus métodos `writeAsStringSync()`, `readAsStringSync()` y `existsSync()` para realizar escritura, lectura y verificación de archivos de forma sincrónica,

El diagrama dinámico que representa el código Archivos es el siguiente:

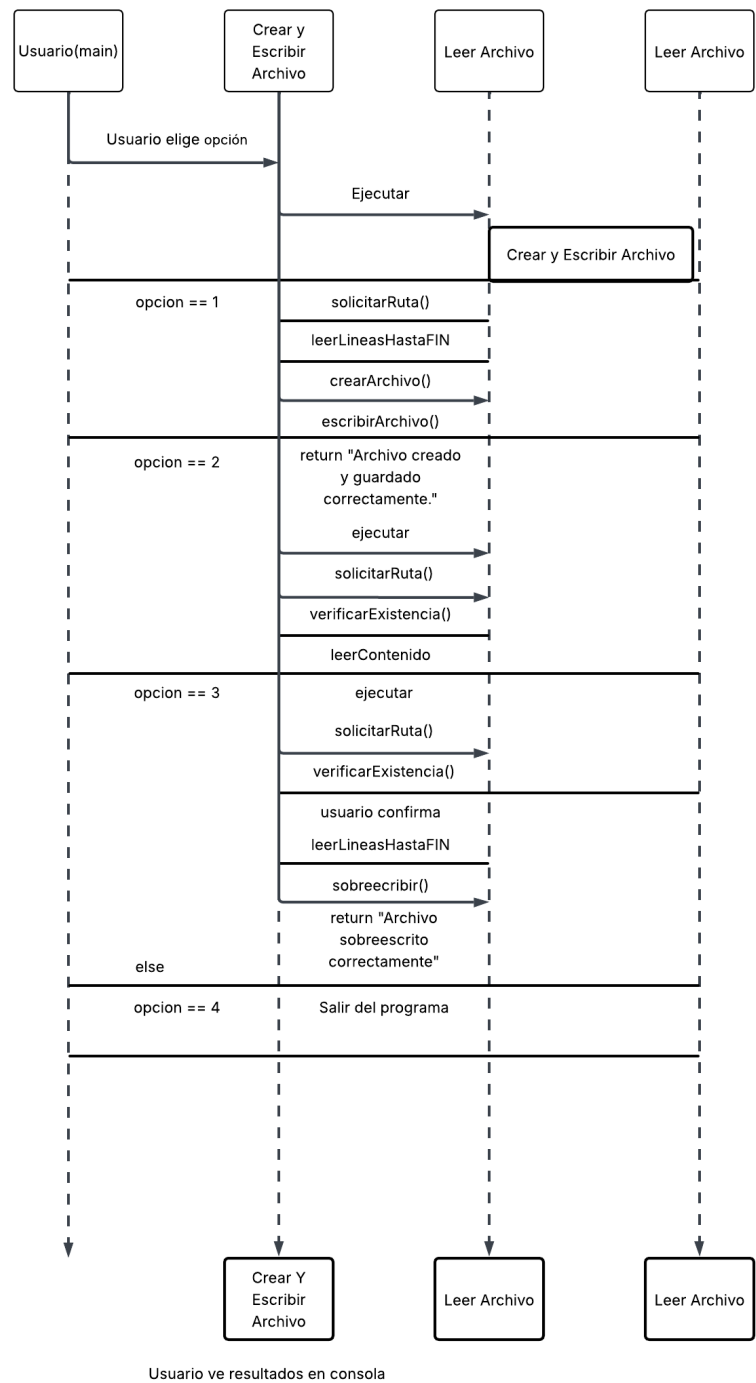


Figura 2: Diagrama dinámico / ARCHIVOS

El diagrama muestra cómo interactúa el usuario con tu programa, y qué métodos se ejecutan dependiendo de la opción que elija en el menú.

Hay 4 participantes:

- Usuario (main) - es quien elige la opción.
- crearYEscribirArchivo()
- leerArchivoExistente()
- sobrescribirArchivo()

Cada uno representa un flujo posible cuando el usuario selecciona del 1 al 4.

3.2. Hilos

4. Resultados

5. Conclusiones

La práctica desarrollada permitió integrar de manera efectiva tres componentes esenciales en el diseño de software: el manejo de archivos como medio de persistencia de datos, la programación concurrente mediante hilos para optimizar el rendimiento, y la aplicación de patrones de diseño como guía estructural para la construcción de sistemas robustos y escalables.

A través del análisis de ejemplos concretos y la elaboración de los diagramas UML estáticos y dinámicos, se logró visualizar con claridad tanto la arquitectura del sistema como su comportamiento en tiempo de ejecución. Esta representación gráfica facilitó la interpretación de los conceptos aplicados en el código, permitiendo identificar relaciones, dependencias y flujos de ejecución que no siempre son evidentes en la lectura directa del programa.

Más allá de la implementación técnica, el enfoque de la práctica se centró en la interpretación crítica de los conceptos, promoviendo una comprensión profunda de las decisiones de diseño y su impacto en la calidad del software. Este ejercicio fortaleció las habilidades de análisis y modelado, pero también fomentó una actitud reflexiva frente al desarrollo, donde cada componente se evalúa en función de su propósito, eficiencia y coherencia dentro del sistema.

Por ende, la práctica contribuyó a consolidar una visión integral del desarrollo de software, articulando teoría y aplicación en un entorno de aprendizaje guiado por ejemplos y representaciones formales.